19 Creating RESTful web services

This chapter covers

- Using ASP.NET Core to create RESTful web services
- Creating web services with the minimal API
- Creating web services with controllers
- Using model binding data from web service requests
- Managing the content produced by web services

Web services accept HTTP requests and generate responses that contain data. In this chapter, I explain how the features provided by the MVC Framework, which is an integral part of ASP.NET Core, can be used to build on the capabilities described in part 2 to create web services.

The nature of web services means that some of the examples in this chapter are tested using command-line tools provided by PowerShell, and it is important to enter the commands exactly as shown. Chapter 20 introduces more sophisticated tools for working with web services, but the command-line approach is better suited to following examples in a book chapter, even if they can feel a little awkward as you type them in. Table 19.1 puts RESTful web services in context.

Table 19.1 Putting RESTful web services in context

Question	Answer
What are they?	Web services provide access to an application's data, typically expressed in the JSON format.
Why are they useful?	Web services are most often used to provide rich client-side applications with data.
How are they used?	The combination of the URL and an HTTP method describes an operation that is handled by an endpoint, or an action method defined by a controller.
Are there any pitfalls or limitations?	There is no widespread agreement about how web services should be implemented, and care must be taken to produce just the data the client expects.
Are there any alternatives?	There are several different approaches to providing clients with data, although RESTful web services are the most common.

Table 19.2 provides a guide to the chapter.

Table 19.2 Chapter guide

Problem	Solution	Listing
Defining a web service	Define endpoints in the `Program.cs` file or create a controller with action methods that correspond to the operations that you require.	3–13
Generating data sequences over time	Use the `IAsyncEnumerable<T>` response, which will prevent the request thread from blocking while results are generated.	14
Preventing request values from being used for sensitive data properties	Use a binding target to restrict the model binding process to only safe properties.	15–17
Expressing nondata outcomes	Use action results to describe the response that ASP.NET Core should send.	18–23
Validating data	Use the ASP.NET Core model binding and model validation features.	24–26
Automatically validating requests	Use the `ApiController` attribute.	27
Omitting null values from data responses	Map the data objects to filter out properties or configure the JSON serializer to ignore `null` properties.	28–32
Apply a rate limit	Use the `EnableRateLimiting` and `DisableRateLimiting`.	33, 34

19.1 Preparing for this chapter

In this chapter, I continue to use the WebApp project created in chapter 18. To prepare for this chapter, drop the database by opening a new PowerShell command prompt, navigating to the folder that contains the `WebApp.csproj` file, and running the command shown in listing 19.1.

TIP You can download the example project for this chapter—and for all the other chapters in this book—from https://github.com/manningbooks/pro-asp.net-core-7. See chapter 1 for how to get help if you have problems running the examples.

Listing 19.1 Dropping the database

```
dotnet ef database drop --force
```

Start the application by running the command shown in listing 19.2 in the project folder.

Listing 19.2 Starting the example application

```
dotnet run
```

Request the URL http://localhost:5000/test once ASP.NET Core has started, and you will see the response shown in figure 19.1.

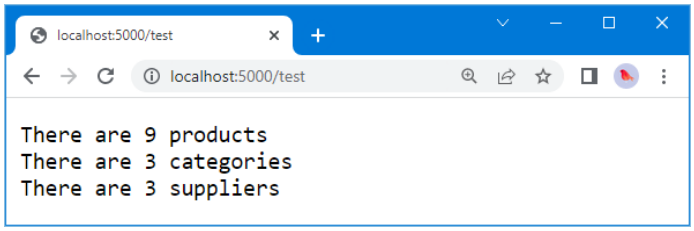

Figure 19.1 Running the example application

19.2 Understanding RESTful web services

Web services respond to HTTP requests with data that can be consumed by clients, such as JavaScript applications. There are no hard-and-fast rules for how web services should work, but the most common approach is to adopt the Representational State Transfer (REST) pattern. There is no authoritative specification for REST, and there is no consensus about what constitutes a RESTful web service, but there are some common themes that are widely used for web services. The lack of a detailed specification leads to endless disagreement about what REST means and how RESTful web services should be created, all of which can be safely ignored if the web services you create work for your projects.

19.2.1 Understanding request URLs and methods

The core premise of REST—and the only aspect for which there is broad agreement—is that a web service defines an API through a combination of URLs and HTTP methods such as GET and POST, which are also known as the HTTP *verbs*. The method specifies the type of operation, while the URL specifies the data object or objects that the operation applies to.

As an example, here is a URL that might identify a `Product` object in the example application:

```
/api/products/1
```

This URL may identify the `Product` object that has a value of `1` for its `ProductId` property. The URL identifies the `Product`, but it is the HTTP method that specifies what should be done with it. Table 19.3 lists the HTTP methods that are commonly used in web services and the operations they conventionally represent.

Table 19.3. HTTP methods and operations

HTTP Method	Description
GET	This method is used to retrieve one or more data objects.
POST	This method is used to create a new object.
PUT	This method is used to update an existing object.
PATCH	This method is used to update part of an existing object.
DELETE	This method is used to delete an object.

19.2.2 Understanding JSON

Most RESTful web services format the response data using the JavaScript Object Notation (JSON) format. JSON has become popular because it is simple and easily consumed by JavaScript clients. JSON is described in detail at www.json.org, but you don't need to understand every aspect of JSON to create web services because ASP.NET Core provides all the features required to create JSON responses.

Understanding the alternatives to RESTful web services

REST isn't the only way to design web services, and there are some popular alternatives. *GraphQL* is most closely associated with the React JavaScript framework, but it can be used more widely. Unlike REST web services, which provide specific queries through individual combinations of a URL and an HTTP method, GraphQL provides access to all an application's data and lets clients query for just the data they require in the format they require. GraphQL can be complex to set up—and can require more sophisticated clients—but the result is a more flexible web service that puts the developers of the client in control of the data they consume. GraphQL isn't supported directly by ASP.NET Core, but there are .NET implementations available. See https://graphql.org for more detail.

A new alternative is gRPC, a full remote procedure call framework that focuses on speed and efficiency. At the time of writing, gRPC cannot be used in web browsers, such as by the Angular or React framework, because browsers don't provide the fine-grained access that gRPC requires to formulate its HTTP requests.

19.3 Creating a web service using the minimal API

As you learn about the facilities that ASP.NET Core provides for web services, it can be easy to forget they are built on the features described in part 2. To create a simple web service, add the statements shown in listing 19.3 to the `Program.cs` file.

Listing 19.3 Creating a web service in the Program.cs file in the Platform folder

```
using Microsoft.EntityFrameworkCore;
using WebApp.Models;
using System.Text.Json;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<DataContext>(opts => {
    opts.UseSqlServer(builder.Configuration[
        "ConnectionStrings:ProductConnection"]);
    opts.EnableSensitiveDataLogging(true);
});

var app = builder.Build();

const string BASEURL = "api/products";

app.MapGet($"{BASEURL}/{{id}}", async (HttpContext context,
        DataContext data) => {
    string? id = context.Request.RouteValues["id"] as string;
    if (id != null) {
        Product? p = data.Products.Find(long.Parse(id));
        if (p == null) {
            context.Response.StatusCode = StatusCodes.Status404NotFound;
        } else {
            context.Response.ContentType = "application/json";
            await context.Response
                .WriteAsync(JsonSerializer.Serialize<Product>(p));
        }
    }
});

app.MapGet(BASEURL, async (HttpContext context, DataContext data) => {
    context.Response.ContentType = "application/json";
    await context.Response.WriteAsync(JsonSerializer
        .Serialize<IEnumerable<Product>>(data.Products));
});

app.MapPost(BASEURL, async (HttpContext context, DataContext data) => {
    Product? p = await
        JsonSerializer.DeserializeAsync<Product>(context.Request.Body);
    if (p != null) {
        await data.AddAsync(p);
       await data.SaveChangesAsync();
        context.Response.StatusCode = StatusCodes.Status200OK;
    }
});
```

```
app.MapGet("/", () => "Hello World!");

var context = app.Services.CreateScope().ServiceProvider
    .GetRequiredService<DataContext>();
SeedData.SeedDatabase(context);

app.Run();
```

The same API that I used to register endpoints in earlier chapters can be used to create a web service, using only features that you have seen before. The `MapGet` and `MapPost` methods are used to create three routes, all of which match URLs that start with `/api`, which is the conventional prefix for web services.

The endpoint for the first route receives a value from a segment variable that is used to locate a single `Product` object in the database. The endpoint for the second route retrieves all the `Product` objects in the database. The third endpoint handles POST requests and reads the request body to get a JSON representation of a new object to add to the database.

There are better ASP.NET Core features for creating web services, which you will see shortly, but the code in listing 19.3 shows how the HTTP method and the URL can be combined to describe an operation, which is the key concept in creating web services.

To test the web service, restart ASP.NET Core and request http://localhost:5000/api/products/1. The request will be matched by the first route defined in listing 19.3 and will produce the response shown on the left of figure 19.2. Next, request http://localhost:5000/api/products, which will be matched by the second route and produce the response shown on the right of figure 19.2.

NOTE The responses shown in the figure contain `null` values for the `Supplier` and `Category` properties because the LINQ queries do not include related data. See chapter 20 for details.

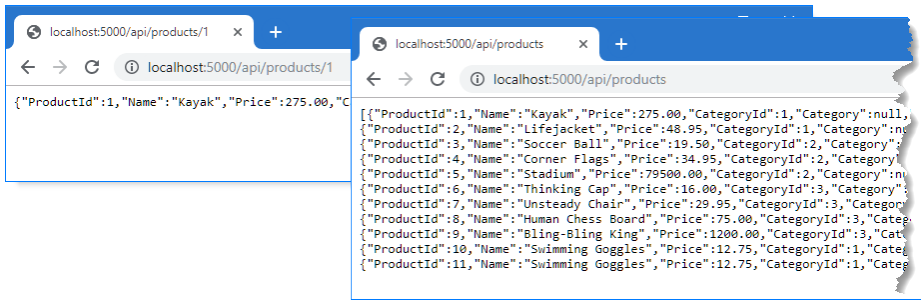

Figure 19.2 Web service response

Testing the third route requires a different approach because it isn't possible to send HTTP POST requests using the browser. Open a new PowerShell command prompt and run the command shown in listing 19.4. It is important to enter the command

exactly as shown because the `Invoke-RestMethod` command is fussy about the syntax of its arguments.

> **TIP** You may receive an error when you use the `Invoke-RestMethod` or `Invoke-WebRequest` command to test the examples in this chapter if you have not performed the initial setup for Microsoft Edge. The problem can be fixed by running the browser and selecting the initial configurations you require.

Listing 19.4 Sending a POST request

```
Invoke-RestMethod http://localhost:5000/api/products -Method POST -Body
➥(@{ Name="Swimming Goggles"; Price=12.75; CategoryId=1; SupplierId=1} |
➥ ConvertTo-Json) -ContentType "application/json"
```

The command sends an HTTP POST command that is matched by the third route defined in listing 19.3. The body of the request is a JSON-formatted object that is parsed to create a `Product`, which is then stored in the database. The JSON object included in the request contains values for the `Name`, `Price`, `CategoryId`, and `SupplierId` properties. The unique key for the object, which is associated with the `ProductId` property, is assigned by the database when the object is stored. Use the browser to request the http://localhost:5000/api/products URL again, and you will see that the JSON response contains the new object, as shown in figure 19.3.

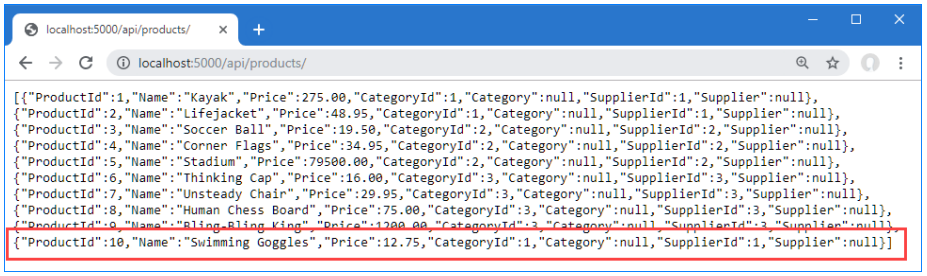

Figure 19.3 Storing new data using the web service

19.4 Creating a web service using a controller

The drawback of using individual endpoints to create a web service is that each endpoint has to duplicate a similar set of steps to produce a response: get the Entity Framework Core service so that it can query the database, set the `Content-Type` header for the response, serialize the objects into JSON, and so on. As a result, web services created with endpoints are difficult to understand and awkward to maintain, and the `Program.cs` file quickly becomes unwieldy.

A more elegant and robust approach is to use a *controller*, which allows a web service to be defined in a single class. Controllers are part of the MVC Framework, which builds on the ASP.NET Core platform and takes care of handling data in the same way that endpoints take care of processing URLs.

The rise and fall of the MVC pattern in ASP.NET Core

The MVC Framework is an implementation of the Model-View-Controller pattern, which describes one way to structure an application. The examples in this chapter use two of the three pillars of the pattern: a data model (the *M* in MVC) and controllers (the *C* in MVC). chapter 21 provides the missing piece and explains how views can be used to create HTML responses using Razor.

The MVC pattern was an important step in the evolution of ASP.NET and allowed the platform to break away from the Web Forms model that predated it. Web Forms applications were easy to start but quickly became difficult to manage and hid details of HTTP requests and responses from the developer. By contrast, the adherence to the MVC pattern provided a strong and scalable structure for applications written with the MVC Framework and hid nothing from the developer. The MVC Framework revitalized ASP.NET and provided the foundation for what became ASP.NET Core, which dropped support for Web Forms and focused solely on using the MVC pattern.

As ASP.NET Core evolved, other styles of web application have been embraced, and the MVC Framework is only one of the ways that applications can be created. That doesn't undermine the utility of the MVC pattern, but it doesn't have the central role that it used to in ASP.NET Core development, and the features that used to be unique to the MVC Framework can now be accessed through other approaches, such as Razor Pages and Blazor.

A consequence of this evolution is that understanding the MVC pattern is no longer a prerequisite for effective ASP.NET Core development. If you are interested in understanding the MVC pattern, then https://en.wikipedia.org/wiki/Model–view–controller is a good place to start. But for this book, understanding how the features provided by the MVC Framework build on the ASP.NET Core platform is all the context that is required.

19.4.1 Enabling the MVC Framework

The first step to creating a web service using a controller is to configure the MVC framework, which requires a service and an endpoint, as shown in listing 19.5. This listing also removes the endpoints defined in the previous section.

Listing 19.5 Enabling the MVC Framework in the Program.cs File in the WebApp folder

```
using Microsoft.EntityFrameworkCore;
using WebApp.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<DataContext>(opts => {
    opts.UseSqlServer(builder.Configuration[
        "ConnectionStrings:ProductConnection"]);
    opts.EnableSensitiveDataLogging(true);
});

builder.Services.AddControllers();

var app = builder.Build();
```

```
app.MapControllers();

app.MapGet("/", () => "Hello World!");

var context = app.Services.CreateScope().ServiceProvider
    .GetRequiredService<DataContext>();
SeedData.SeedDatabase(context);

app.Run();
```

The `AddControllers` method defines the services that are required by the MVC framework, and the `MapControllers` method defines routes that will allow controllers to handle requests. You will see other methods used to configure the MVC framework used in later chapters, which provide access to different features, but the methods used in listing 19.5 are the ones that configure the MVC framework for web services.

19.4.2 Creating a controller

Controllers are classes whose methods, known as *actions*, can process HTTP requests. Controllers are discovered automatically when the application is started. The basic discovery process is simple: any public class whose name ends with `Controller` is a controller, and any `public` method a controller defines is an action. To demonstrate how simple a controller can be, create the `WebApp/Controllers` folder and add to it a file named `ProductsController.cs` with the code shown in listing 19.6.

TIP Controllers are conventionally defined in the `Controllers` folder, but they can be defined anywhere in the project and will still be discovered.

Listing 19.6 The contents of the ProductsController.cs file in the Controllers folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;

namespace WebApp.Controllers {

    [Route("api/[controller]")]
    public class ProductsController : ControllerBase {

        [HttpGet]
        public IEnumerable<Product> GetProducts() {
            return new Product[] {
                new Product() { Name = "Product #1" },
                new Product() { Name = "Product #2" },
            };
        }

        [HttpGet("{id}")]
        public Product GetProduct() {
            return new Product() {
                ProductId = 1, Name = "Test Product"
            };
        }
    }
}
```

The `ProductsController` class meets the criteria that the MVC framework looks for in a controller. It defines public methods named `GetProducts` and `GetProduct`, which will be treated as actions.

Understanding the base class

Controllers are derived from the `ControllerBase` class, which provides access to features provided by the MVC Framework and the underlying ASP.NET Core platform. Table 19.4 describes the most useful properties provided by the `ControllerBase` class.

> **NOTE** Although controllers are typically derived from the `ControllerBase` or `Controller` classes (described in chapter 21), this is just convention, and the MVC Framework will accept any class whose name ends with `Controller`, that is derived from a class whose name ends with `Controller`, or that has been decorated with the `Controller` attribute. Apply the `NonController` attribute to classes that meet these criteria but that should not receive HTTP requests.

Table 19.4 Useful ControllerBase properties

Name	Description
`HttpContext`	This property returns the `HttpContext` object for the current request.
`ModelState`	This property returns details of the data validation process, as demonstrated in the "Validating Data" section later in the chapter and described in detail in chapter 29.
`Request`	This property returns the `HttpRequest` object for the current request.
`Response`	This property returns the `HttpResponse` object for the current response.
`RouteData`	This property returns the data extracted from the request URL by the routing middleware, as described in chapter 13.
`User`	This property returns an object that describes the user associated with the current request, as described in chapter 38.

A new instance of the controller class is created each time one of its actions is used to handle a request, which means the properties in table 19.4 describe only the current request.

Understanding the controller attributes

The HTTP methods and URLs supported by the action methods are determined by the combination of attributes that are applied to the controller. The URL for the controller is specified by the `Route` attribute, which is applied to the class, like this:

```
...
[Route("api/[controller]")]
public class ProductsController: ControllerBase {
...
```

The `[controller]` part of the attribute argument is used to derive the URL from the name of the controller class. The `Controller` part of the class name is dropped, which means that the attribute in listing 19.6 sets the URL for the controller to `/api/products`.

Each action is decorated with an attribute that specifies the HTTP method that it supports, like this:

```
...
[HttpGet]
public Product[] GetProducts() {
...
```

The name given to action methods doesn't matter in controllers used for web services. There are other uses for controllers, described in chapter 21, where the name does matter, but for web services, it is the HTTP method attributes and the route patterns that are important.

The `HttpGet` attribute tells the MVC framework that the `GetProducts` action method will handle HTTP GET requests. Table 19.5 describes the full set of attributes that can be applied to actions to specify HTTP methods.

Table 19.5 The HTTP method attributes

Name	Description
`HttpGet`	This attribute specifies that the action can be invoked only by HTTP requests that use the GET verb.
`HttpPost`	This attribute specifies that the action can be invoked only by HTTP requests that use the POST verb.
`HttpDelete`	This attribute specifies that the action can be invoked only by HTTP requests that use the DELETE verb.
`HttpPut`	This attribute specifies that the action can be invoked only by HTTP requests that use the PUT verb.
`HttpPatch`	This attribute specifies that the action can be invoked only by HTTP requests that use the PATCH verb.
`HttpHead`	This attribute specifies that the action can be invoked only by HTTP requests that use the HEAD verb.
`AcceptVerbs`	This attribute is used to specify multiple HTTP verbs.

The attributes applied to actions to specify HTTP methods can also be used to build on the controller's base URL.

```
...
[HttpGet("{id}")]
public Product GetProduct() {
...
```

This attribute tells the MVC framework that the `GetProduct` action method handles GET requests for the URL pattern `api/products/{id}`. During the discovery process, the attributes applied to the controller are used to build the set of URL patterns that the controller can handle, summarized in table 19.6.

TIP When writing a controller, it is important to ensure that each combination of the HTTP method and URL pattern that the controller supports is mapped to only one action method. An exception will be thrown when a request can be handled by multiple actions because the MVC Framework is unable to decide which to use.

Table 19.6 The URL patterns

HTTP Method	URL Pattern	Action Method Name
GET	`api/products`	`GetProducts`
GET	`api/products/{id}`	`GetProduct`

You can see how the combination of attributes is equivalent to the `MapGet` methods I used for the same URL patterns when I used endpoints to create a web service earlier in the chapter.

GET and POST: Pick the right one

The rule of thumb is that GET requests should be used for all read-only information retrieval, while POST requests should be used for any operation that changes the application state. In standards-compliance terms, GET requests are for *safe* interactions (having no side effects besides information retrieval), and POST requests are for *unsafe* interactions (making a decision or changing something). These conventions are set by the World Wide Web Consortium (W3C), at https://www.rfc-editor.org/rfc/rfc9110.html.

GET requests are *addressable*: all the information is contained in the URL, so it's possible to bookmark and link to these addresses. Do not use GET requests for operations that change state. Many web developers learned this the hard way in 2005 when Google Web Accelerator was released to the public. This application prefetched all the content linked from each page, which is legal within the HTTP because GET requests should be safe. Unfortunately, many web developers had ignored the HTTP conventions and placed simple links to "delete item" or "add to shopping cart" in their applications. Chaos ensued.

UNDERSTANDING ACTION METHOD RESULTS

One of the main benefits provided by controllers is that the MVC Framework takes care of setting the response headers and serializing the data objects that are sent to the client. You can see this in the results defined by the action methods, like this:

```
...
[HttpGet("{id}")]
public Product GetProduct() {
...
```

When I used an endpoint, I had to work directly with the JSON serializer to create a string that can be written to the response and set the `Content-Type` header to tell the client that the response contained JSON data. The action method returns a `Product` object, which is processed automatically.

To see how the results from the action methods are handled, restart ASP.NET Core and request http://localhost:5000/api/products, which will produce the response shown on the left of figure 19.4, which is produced by the `GetProducts` action method. Next, request http://localhost:5000/api/products/1, which will be handled by the `GetProduct` method and produce the result shown on the right side of figure 19.4.

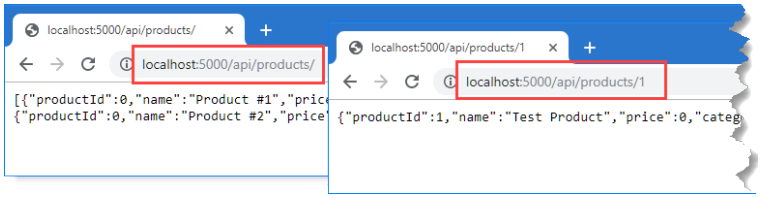

Figure 19.4 Using a controller

USING DEPENDENCY INJECTION IN CONTROLLERS

A new instance of the controller class is created each time one of its actions is used to handle a request. The application's services are used to resolve any dependencies the controller declares through its constructor and any dependencies that the action method defines. This allows services that are required by all actions to be handled through the constructor while still allowing individual actions to declare their own dependencies, as shown in listing 19.7.

Listing 19.7 Using services in the ProductsController.cs file in the Controllers folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;

namespace WebApp.Controllers {

    [Route("api/[controller]")]
    public class ProductsController : ControllerBase {
        private DataContext context;

        public ProductsController(DataContext ctx) {
            context = ctx;
        }

        [HttpGet]
        public IEnumerable<Product> GetProducts() {
            return context.Products;
        }

        [HttpGet("{id}")]
        public Product? GetProduct([FromServices]
                ILogger<ProductsController> logger) {
            logger.LogInformation("GetProduct Action Invoked");
            return context.Products
                .OrderBy(p => p.ProductId).FirstOrDefault();
        }
    }
}
```

The constructor declares a dependency on the `DataContext` service, which provides access to the application's data. The services are resolved using the request scope, which means that a controller can request all services, without needing to understand their lifecycle.

The Entity Framework Core context service lifecycle

A new Entity Framework Core context object is created for each controller. Some developers will try to reuse context objects as a perceived performance improvement, but this causes problems because data from one query can affect subsequent queries. Behind the scenes, Entity Framework Core efficiently manages the connections to the database, and you should not try to store or reuse context objects outside of the controller for which they are created.

The `GetProducts` action method uses the `DataContext` to request all the `Product` objects in the database. The `GetProduct` method also uses the `DataContext` service, but it declares a dependency on `ILogger<T>`, which is the logging service described in chapter 15. Dependencies that are declared by action methods must be decorated with the `FromServices` attribute, like this:

```
...
public Product GetProduct([FromServices]
    ILogger<ProductsController> logger)
...
```

By default, the MVC Framework attempts to find values for action method parameters from the request URL, and the `FromServices` attribute overrides this behavior. The `FromServices` attribute can often be omitted, and ASP.NET Core will try to resolve parameters using dependency injection, but this doesn't work for all parameter types, and I prefer to use the attribute to clearly denote that the value for the parameter will be provided by dependency injection.

To see the use of the services in the controller, restart ASP.NET Core and request http://localhost:5000/api/products/1, which will produce the response shown in figure 19.5. You will also see the following logging message in the application's output:

```
...
info: WebApp.Controllers.ProductsController[0]
      GetProduct Action Invoked
...
```

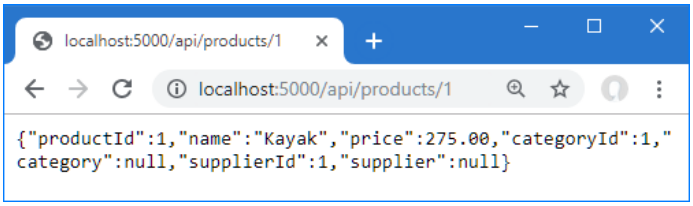

Figure 19.5 Using services in a controller

CAUTION One consequence of the controller lifecycle is that you can't rely on side effects caused by methods being called in a specific sequence. So, for example, I can't assign the `ILogger<T>` object received by the `GetProduct` method in listing 19.7 to a property that can be read by the `GetProducts` action in later requests. Each controller object is used to handle one request, and only one action method will be invoked by the MVC Framework for each object.

USING MODEL BINDING TO ACCESS ROUTE DATA

In the previous section, I noted that the MVC Framework uses the request URL to find values for action method parameters, a process known as *model binding*. Model binding is described in detail in chapter 28, but listing 19.8 shows a simple example.

Listing 19.8 Model binding in the ProductsController.cs file in the Controllers folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;

namespace WebApp.Controllers {

    [Route("api/[controller]")]
    public class ProductsController : ControllerBase {
        private DataContext context;

        public ProductsController(DataContext ctx) {
            context = ctx;
        }

        [HttpGet]
        public IEnumerable<Product> GetProducts() {
            return context.Products;
        }

        [HttpGet("{id}")]
        public Product? GetProduct(long id,
                [FromServices] ILogger<ProductsController> logger) {
            logger.LogDebug("GetProduct Action Invoked");
            return context.Products.Find(id);
        }
    }
}
```

The listing adds a `long` parameter named `id` to the `GetProduct` method. When the action method is invoked, the MVC Framework injects the value with the same name from the routing data, automatically converting it to a `long` value, which is used by the action to query the database using the LINQ `Find` method. The result is that the action method responds to the URL, which you can see by restarting ASP.NET Core and requesting http://localhost:5000/api/products/5, which will produce the response shown in figure 19.6.

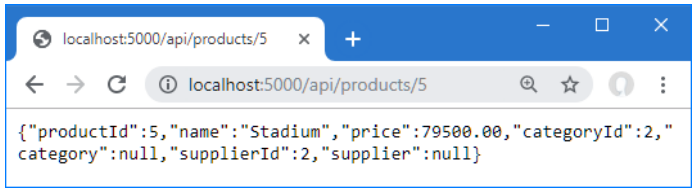

Figure 19.6 Using model binding in an action

Model binding from the request body

The model binding feature can also be used on the data in the request body, which allows clients to send data that is easily received by an action method. Listing 19.9 adds a new action method that responds to POST requests and allows clients to provide a JSON representation of the `Product` object in the request body.

Listing 19.9 Adding an action in the ProductsController.cs file in the Controllers folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;

namespace WebApp.Controllers {

    [Route("api/[controller]")]
    public class ProductsController : ControllerBase {
        private DataContext context;

        public ProductsController(DataContext ctx) {
            context = ctx;
        }

        [HttpGet]
        public IEnumerable<Product> GetProducts() {
            return context.Products;
        }

        [HttpGet("{id}")]
        public Product? GetProduct(long id,
                [FromServices] ILogger<ProductsController> logger) {
            logger.LogDebug("GetProduct Action Invoked");
            return context.Products.Find(id);
        }

        [HttpPost]
        public void SaveProduct([FromBody] Product product) {
            context.Products.Add(product);
            context.SaveChanges();
        }
    }
}
```

The new action relies on two attributes. The `HttpPost` attribute is applied to the action method and tells the MVC Framework that the action can process POST requests. The `FromBody` attribute is applied to the action's parameter, and it specifies that the value

for this parameter should be obtained by parsing the request body. When the action method is invoked, the MVC Framework will create a new `Product` object and populate its properties with the values in the request body. The model binding process can be complex and is usually combined with data validation, as described in chapter 29, but for a simple demonstration, restart ASP.NET Core, open a new PowerShell command prompt, and run the command shown in listing 19.10.

Listing 19.10 Sending a POST request to the example application

```
Invoke-RestMethod http://localhost:5000/api/products -Method POST -Body  (@
➥{ Name="Soccer Boots"; Price=89.99; CategoryId=2; SupplierId=2} |
➥ ConvertTo-Json) -ContentType "application/json"
```

Once the command has executed, use a web browser to request http://localhost:5000/api/products, and you will see the new object that has been stored in the database, as shown in figure 19.7.

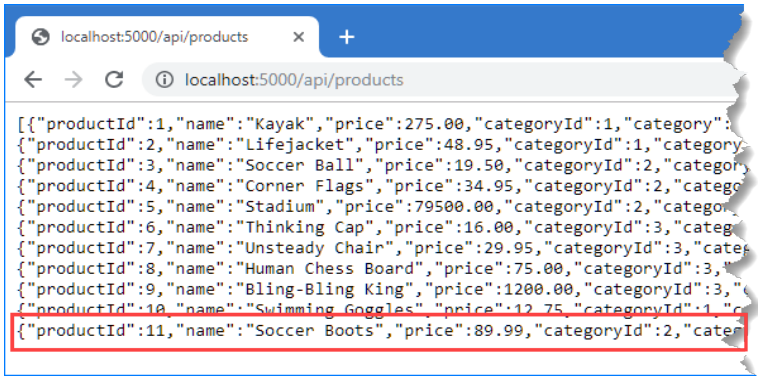

Figure 19.7 Storing new data using a controller

ADDING ADDITIONAL ACTIONS

Now that the basic features are in place, I can add actions that allow clients to replace and delete `Product` objects using the HTTP PUT and DELETE methods, as shown in listing 19.11.

Listing 19.11 Adding actions in the ProductsController.cs file in the Controllers folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;

namespace WebApp.Controllers {

    [Route("api/[controller]")]
    public class ProductsController : ControllerBase {
        private DataContext context;

        public ProductsController(DataContext ctx) {
            context = ctx;
        }
```

```
        [HttpGet]
        public IEnumerable<Product> GetProducts() {
            return context.Products;
        }

        [HttpGet("{id}")]
        public Product? GetProduct(long id,
                [FromServices] ILogger<ProductsController> logger) {
            logger.LogDebug("GetProduct Action Invoked");
            return context.Products.Find(id);
        }

        [HttpPost]
        public void SaveProduct([FromBody] Product product) {
            context.Products.Add(product);
            context.SaveChanges();
        }

        [HttpPut]
        public void UpdateProduct([FromBody] Product product) {
            context.Products.Update(product);
            context.SaveChanges();
        }

        [HttpDelete("{id}")]
        public void DeleteProduct(long id) {
            context.Products.Remove(new Product() {
                ProductId = id, Name = string.Empty
            });
            context.SaveChanges();
        }
    }
}
```

The `UpdateProduct` action is similar to the `SaveProduct` action and uses model binding to receive a `Product` object from the request body. The `DeleteProduct` action receives a primary key value from the URL and uses it to create a `Product` that has a value for the `ProductId` property, which is required because Entity Framework Core works only with objects, but web service clients typically expect to be able to delete objects using just a key value. (The empty string is assigned to the `Name` property, to which the `required` keyword has been applied and without which a `Product` object cannot be created. Entity Framework Core ignores the empty string when identifying the data to delete).

Restart ASP.NET Core and then use a different PowerShell command prompt to run the command shown in listing 19.12, which tests the `UpdateProduct` action.

Listing 19.12 Updating an object

```
Invoke-RestMethod http://localhost:5000/api/products -Method PUT -Body  (@{
ProductId=1; Name="Green Kayak"; Price=275; CategoryId=1; SupplierId=1} |
ConvertTo-Json) -ContentType "application/json"
```

The command sends an HTTP PUT request whose body contains a replacement object. The action method receives the object through the model binding feature

and updates the database. Next, run the command shown in listing 19.13 to test the `DeleteProduct` action.

Listing 19.13 Deleting an object

```
Invoke-RestMethod http://localhost:5000/api/products/2 -Method DELETE
```

This command sends an HTTP DELETE request, which will delete the object whose `ProductId` property is 2. To see the effect of the changes, use the browser to request http://localhost:5000/api/products, which will send a GET request that is handled by the `GetProducts` action and produce the response shown in figure 19.8.

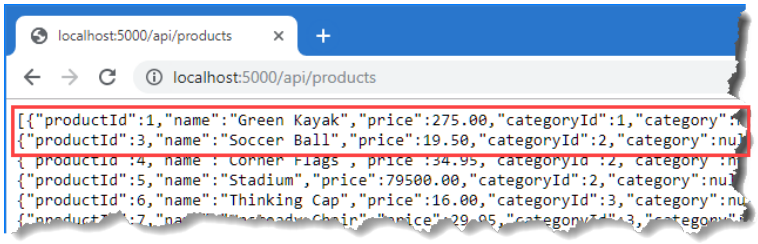

Figure 19.8 Updating and deleting objects

19.5 Improving the web service

The controller in listing 19.11 re-creates the functionality provided by the separate endpoints, but there are still improvements that can be made, as described in the following sections.

Supporting cross-origin requests

If you are supporting third-party JavaScript clients, you may need to enable support for cross-origin requests (CORS). Browsers protect users by only allowing JavaScript code to make HTTP requests within the same origin, which means to URLs that have the same scheme, host, and port as the URL used to load the JavaScript code. CORS loosens this restriction by performing an initial HTTP request to check that the server will allow requests originating from a specific URL, helping prevent malicious code using your service without the user's consent.

ASP.NET Core provides a built-in service that handles CORS, which is enabled by adding the following statement to the `Program.cs` file:

```
...
builder.Services.AddCors();
...
```

The options pattern is used to configure CORS with the `CorsOptions` class defined in the `Microsoft.AspNetCore.Cors.Infrastructure` namespace. See https://docs.microsoft.com/en-gb/aspnet/core/security/cors for details.

19.5.1 Using asynchronous actions

The ASP.NET Core platform processes each request by assigning a thread from a pool. The number of requests that can be processed concurrently is limited to the size of the pool, and a thread can't be used to process any other request while it is waiting for an action to produce a result.

Actions that depend on external resources can cause a request thread to wait for an extended period. A database server, for example, may have its own concurrency limits and may queue up queries until they can be executed. The ASP.NET Core request thread is unavailable to process any other requests until the database produces a result for the action, which then produces a response that can be sent to the HTTP client.

This problem can be addressed by defining asynchronous actions, which allow ASP.NET Core threads to process other requests when they would otherwise be blocked, increasing the number of HTTP requests that the application can process simultaneously. Listing 19.14 revises the controller to use asynchronous actions.

NOTE Asynchronous actions don't produce responses any quicker, and the benefit is only to increase the number of requests that can be processed concurrently.

Listing 19.14 Async actions in the ProductsController.cs file in the Controllers folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;

namespace WebApp.Controllers {

    [Route("api/[controller]")]
    public class ProductsController : ControllerBase {
        private DataContext context;

        public ProductsController(DataContext ctx) {
            context = ctx;
        }

        [HttpGet]
        public IAsyncEnumerable<Product> GetProducts() {
            return context.Products.AsAsyncEnumerable();
        }

        [HttpGet("{id}")]
        public async Task<Product?> GetProduct(long id) {
            return await context.Products.FindAsync(id);
        }

        [HttpPost]
        public async Task SaveProduct([FromBody] Product product) {
            await context.Products.AddAsync(product);
            await context.SaveChangesAsync();
        }

        [HttpPut]
        public async Task UpdateProduct([FromBody] Product product) {
```

```
            context.Update(product);
            await context.SaveChangesAsync();
        }

        [HttpDelete("{id}")]
        public async Task DeleteProduct(long id) {
            context.Products.Remove(new Product() {
                ProductId = id, Name = string.Empty
            });
            await context.SaveChangesAsync();
        }
    }
}
```

Entity Framework Core provides asynchronous versions of some methods, such as `FindAsync`, `AddAsync`, and `SaveChangesAsync`, and I have used these with the `await` keyword. Not all operations can be performed asynchronously, which is why the `Update` and `Remove` methods are unchanged within the `UpdateProduct` and `DeleteProduct` actions.

For some operations—including LINQ queries to the database—the `IAsyncEnumerable<T>` interface can be used, which denotes a sequence of objects that should be enumerated asynchronously and prevents the ASP.NET Core request thread from waiting for each object to be produced by the database, as explained in chapter 5.

There is no change to the responses produced by the controller, but the threads that ASP.NET Core assigns to process each request are not necessarily blocked by the action methods.

19.5.2 Preventing over-binding

Some of the action methods use the model binding feature to get data from the request body so that it can be used to perform database operations. There is a problem with the `SaveProduct` action, which can be seen by using a PowerShell prompt to run the command shown in listing 19.15.

Listing 19.15 Saving a product

```
Invoke-RestMethod http://localhost:5000/api/products -Method POST -Body  (@
➥{ ProductId=100; Name="Swim Buoy"; Price=19.99; CategoryId=1;
➥SupplierId=1} | ConvertTo-Json) -ContentType "application/json"
```

This command fails with an error. Unlike the command that was used to test the POST method, this command includes a value for the `ProductId` property. When Entity Framework Core sends the data to the database, the following exception is thrown:

```
...
Microsoft.Data.SqlClient.SqlException (0x80131904): Cannot insert explicit
value for identity column in table 'Products' when IDENTITY_INSERT
is set to OFF.
...
```

By default, Entity Framework Core configures the database to assign primary key values when new objects are stored. This means the application doesn't have to worry about keeping track of which key values have already been assigned and allows multiple

applications to share the same database without the need to coordinate key allocation. The `Product` data model class needs a `ProductId` property, but the model binding process doesn't understand the significance of the property and adds any values that the client provides to the objects it creates, which causes the exception in the `SaveProduct` action method.

This is known as *over-binding*, and it can cause serious problems when a client provides values that the developer didn't expect. At best, the application will behave unexpectedly, but this technique has been used to subvert application security and grant users more access than they should have.

The safest way to prevent over-binding is to create separate data model classes that are used only for receiving data through the model binding process. Add a class file named `ProductBindingTarget.cs` to the `WebApp/Models` folder and use it to define the class shown in listing 19.16.

Listing 19.16 The ProductBindingTarget.cs file in the WebApp/Models folder

```
namespace WebApp.Models {
    public class ProductBindingTarget {

        public required string Name { get; set; }

        public decimal Price { get; set; }

        public long CategoryId { get; set; }

        public long SupplierId { get; set; }

        public Product ToProduct() => new Product() {
            Name = this.Name, Price = this.Price,
            CategoryId = this.CategoryId, SupplierId = this.SupplierId
        };
    }
}
```

The `ProductBindingTarget` class defines only the properties that the application wants to receive from the client when storing a new object. The `ToProduct` method creates a `Product` that can be used with the rest of the application, ensuring that the client can provide properties only for the `Name`, `Price`, `CategoryId`, and `SupplierId` properties. Listing 19.17 uses the binding target class in the `SaveProduct` action to prevent over-binding.

Listing 19.17 A binding target in the ProductsController.cs file in the Controllers folder

```
...
[HttpPost]
public async Task SaveProduct([FromBody] ProductBindingTarget target) {
    await context.Products.AddAsync(target.ToProduct());
    await context.SaveChangesAsync();
}
...
```

Restart ASP.NET Core and repeat the command from listing 19.15, and you will see the response shown in figure 19.9. The client has included the `ProductId` value, but it is ignored by the model binding process, which discards values for read-only properties. (You may see a different value for the `ProductId` property when you run this example depending on the changes you made to the database before running the command.)

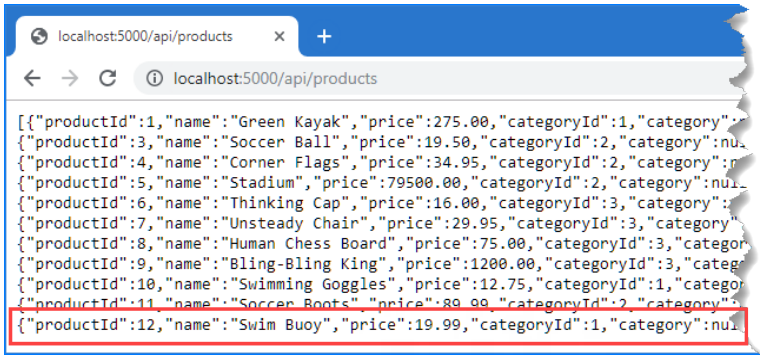

Figure 19.9 Discarding unwanted data values

19.5.3 Using action results

ASP.NET Core sets the status code for responses automatically, but you won't always get the result you desire, in part because there are no firm rules for RESTful web services, and the assumptions that Microsoft makes may not match your expectations. To see an example, use a PowerShell command prompt to run the command shown in listing 19.18, which sends a GET request to the web service.

Listing 19.18 Sending a GET request

```
Invoke-WebRequest http://localhost:5000/api/products/1000 | Select-Object
➥StatusCode
```

The `Invoke-WebRequest` command is similar to the `Invoke-RestMethod` command used in earlier examples but makes it easier to get the status code from the response. The URL requested in listing 19.18 will be handled by the `GetProduct` action method, which will query the database for an object whose `ProductId` value is `1000`, and the command produces the following output:

```
StatusCode
----------
       204
```

There is no matching object in the database, which means that the `GetProduct` action method returns `null`. When the MVC Framework receives `null` from an action method, it returns the 204 status code, which indicates a successful request that has produced no data. Not all web services behave this way, and a common alternative is to return a 404 response, indicating not found.

Similarly, the `SaveProducts` action will return a 200 response when it stores an object, but since the primary key isn't generated until the data is stored, the client doesn't know what key value was assigned.

NOTE There is no right or wrong when it comes to these kinds of web service implementation details, and you should pick the approaches that best suit your project and personal preferences. This section is an example of how to change the default behavior and not a direction to follow any specific style of web service.

Action methods can direct the MVC Framework to send a specific response by returning an object that implements the `IActionResult` interface, which is known as an *action result*. This allows the action method to specify the type of response that is required without having to produce it directly using the `HttpResponse` object.

The `ControllerBase` class provides a set of methods that are used to create action result objects, which can be returned from action methods. Table 19.7 describes the most useful action result methods.

Table 19.7 Useful ControllerBase action result methods

Name	Description
`Ok`	The `IActionResult` returned by this method produces a 200 OK status code and sends an optional data object in the response body.
`NoContent`	The `IActionResult` returned by this method produces a 204 NO CONTENT status code.
`BadRequest`	The `IActionResult` returned by this method produces a 400 BAD REQUEST status code. The method accepts an optional model state object that describes the problem to the client, as demonstrated in the "Validating Data" section.
`File`	The `IActionResult` returned by this method produces a 200 OK response, sets the `Content-Type` header to the specified type, and sends the specified file to the client.
`NotFound`	The `IActionResult` returned by this method produces a 404 NOT FOUND status code.
`Redirect` `RedirectPermanent`	The `IActionResult` returned by these methods redirects the client to a specified URL.
`RedirectToRoute` `RedirectToRoutePermanent`	The `IActionResult` returned by these methods redirects the client to the specified URL that is created using the routing system, using convention routing, as described in the "Redirecting Using Route Values" sidebar.
`LocalRedirect` `LocalRedirectPermanent`	The `IActionResult` returned by these methods redirects the client to the specified URL that is local to the application.
`RedirectToAction` `RedirectToActionPermanent`	The `IActionResult` returned by these methods redirects the client to an action method. The URL for the redirection is created using the URL routing system.
`RedirectToPage` `RedirectToPagePermanent`	The `IActionResult` returned by these methods redirects the client to a Razor Page, described in chapter 23.
`StatusCode`	The `IActionResult` returned by this method produces a response with a specific status code.

When an action method returns an object, it is equivalent to passing the object to the `Ok` method and returning the result. When an action returns `null`, it is equivalent to returning the result from the `NoContent` method. Listing 19.19 revises the behavior of the `GetProduct` and `SaveProduct` actions so they use the methods from table 19.7 to override the default behavior for web service controllers.

Listing 19.19 Action results in the ProductsController.cs file in the Controllers folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;

namespace WebApp.Controllers {

    [Route("api/[controller]")]
    public class ProductsController : ControllerBase {
        private DataContext context;

        public ProductsController(DataContext ctx) {
            context = ctx;
        }

        [HttpGet]
        public IAsyncEnumerable<Product> GetProducts() {
            return context.Products.AsAsyncEnumerable();
        }

        [HttpGet("{id}")]
        public async Task<IActionResult> GetProduct(long id) {
            Product? p = await context.Products.FindAsync(id);
            if (p == null) {
                return NotFound();
            }
            return Ok(p);
        }

        [HttpPost]
        public async Task<IActionResult>
                SaveProduct([FromBody] ProductBindingTarget target) {
            Product p = target.ToProduct();
            await context.Products.AddAsync(p);
            await context.SaveChangesAsync();
            return Ok(p);
        }

        [HttpPut]
        public async Task UpdateProduct([FromBody] Product product) {
            context.Update(product);
            await context.SaveChangesAsync();
        }

        [HttpDelete("{id}")]
        public async Task DeleteProduct(long id) {
            context.Products.Remove(new Product() {
                ProductId = id, Name = string.Empty
```

```
            });
            await context.SaveChangesAsync();
        }
    }
}
```

Restart ASP.NET Core and repeat the command from listing 19.18, and you will see an exception, which is how the `Invoke-WebRequest` command responds to error status codes, such as the 404 Not Found returned by the `GetProduct` action method.

To see the effect of the change to the `SaveProduct` action method, use a PowerShell command prompt to run the command shown in listing 19.20, which sends a POST request to the web service.

Listing 19.20 Sending a POST request

```
Invoke-RestMethod http://localhost:5000/api/products -Method POST -Body
➥(@{Name="Boot Laces"; Price=19.99; CategoryId=2; SupplierId=2} |
➥ConvertTo-Json) -ContentType "application/json"
```

The command will produce the following output, showing the values that were parsed from the JSON data received from the web service:

```
productId  : 13
name       : Boot Laces
price      : 19.99
categoryId : 2
category   :
supplierId : 2
supplier   :
```

Performing redirections

Many of the action result methods in table 19.7 relate to redirections, which redirect the client to another URL. The most basic way to perform a redirection is to call the `Redirect` method, as shown in listing 19.21.

> **TIP** The `LocalRedirect` and `LocalRedirectPermanent` methods throw an exception if a controller tries to perform a redirection to any URL that is not local. This is useful when you are redirecting to URLs provided by users, where an *open redirection attack* is attempted to redirect another user to an untrusted site.

Listing 19.21 Redirecting in the ProductsController.cs file in the Controllers folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;

namespace WebApp.Controllers {

    [Route("api/[controller]")]
    public class ProductsController : ControllerBase {
        private DataContext context;

        public ProductsController(DataContext ctx) {
```

```
            context = ctx;
        }

        // ...other action methods omitted for brevity...

        [HttpGet("redirect")]
        public IActionResult Redirect() {
            return Redirect("/api/products/1");
        }
    }
}
```

The redirection URL is expressed as a `string` argument to the `Redirect` method, which produces a temporary redirection. Restart ASP.NET Core and use a PowerShell command prompt to run the command shown in listing 19.22, which sends a GET request that will be handled by the new action method.

Listing 19.22 Testing redirection

```
Invoke-RestMethod http://localhost:5000/api/products/redirect
```

The `Invoke-RestMethod` command will receive the redirection response from the web service and send a new request to the URL it is given, producing the following response:

```
productId  : 1
name       : Green Kayak
price      : 275.00
categoryId : 1
category   :
supplierId : 1
supplier   :
```

REDIRECTING TO AN ACTION METHOD

You can redirect to another action method using the `RedirectToAction` method (for temporary redirections) or the `RedirectToActionPermanent` method (for permanent redirections). Listing 19.23 changes the `Redirect` action method so that the client will be redirected to another action method defined by the controller.

Listing 19.23 Redirecting in the ProductsController.cs file in the Controllers folder

```
...
[HttpGet("redirect")]
public IActionResult Redirect() {
    return RedirectToAction(nameof(GetProduct), new { Id = 1 });
}
...
```

The action method is specified as a string, although the `nameof` expression can be used to select an action method without the risk of a typo. Any additional values required to create the route are supplied using an anonymous object. Restart ASP.NET Core and use a PowerShell command prompt to repeat the command in listing 19.22. The routing system will be used to create a URL that targets the specified action method, producing the following response:

```
productId  : 1
name       : Green Kayak
price      : 275.00
categoryId : 1
category   :
supplierId : 1
supplier   :
```

If you specify only an action method name, then the redirection will target the current controller. There is an overload of the `RedirectToAction` method that accepts action and controller names.

Redirecting using route values

The `RedirectToRoute` and `RedirectToRoutePermanent` methods redirect the client to a URL that is created by providing the routing system with values for segment variables and allowing it to select a route to use. This can be useful for applications with complex routing configurations, and caution should be used because it is easy to create a redirection to the wrong URL. Here is an example of redirection with the `RedirectToRoute` method:

```
...
[HttpGet("redirect")]
public IActionResult Redirect() {
   return RedirectToRoute(new {
      controller = "Products", action = "GetProduct", Id = 1
   });
}
...
```

The set of values in this redirection relies on convention routing to select the controller and action method. Convention routing is typically used with controllers that produce HTML responses, as described in chapter 21.

19.5.4 Validating data

When you accept data from clients, you must assume that a lot of the data will be invalid and be prepared to filter out values that the application can't use. The data validation features provided for MVC Framework controllers are described in detail in chapter 29, but for this chapter, I am going to focus on only one problem: ensuring that the client provides values for the properties that are required to store data in the database. The first step in model binding is to apply attributes to the properties of the data model class, as shown in listing 19.24.

Listing 19.24 Attributes in the ProductBindingTarget.cs file in the Models folder

```
using System.ComponentModel.DataAnnotations;

namespace WebApp.Models {
    public class ProductBindingTarget {
```

```
        [Required]
        public required string Name { get; set; }

        [Range(1, 1000)]
        public decimal Price { get; set; }

        [Range(1, long.MaxValue)]
        public long CategoryId { get; set; }

        [Range(1, long.MaxValue)]
        public long SupplierId { get; set; }

        public Product ToProduct() => new Product() {
            Name = this.Name, Price = this.Price,
            CategoryId = this.CategoryId, SupplierId = this.SupplierId
        };
    }
}
```

The `Required` attribute denotes properties for which the client must provide a value and can be applied to properties that are assigned `null` when there is no value in the request. The `Range` attribute requires a value between upper and lower limits and is used for primitive types that will default to zero when there is no value in the request.

> **NOTE** The `Required` attribute could be omitted from the `Name` property because ASP.NET Core will infer the validation constraint from the `required` keyword. This is a useful feature, but I like to use the `Required` attribute for consistency and to make it obvious that the validation constraint was intentional.

Listing 19.25 updates the `SaveProduct` action to perform validation before storing the object that is created by the model binding process, ensuring that only objects that contain values for all four properties decorated with the validation attributes are accepted.

Listing 19.25 Validation in the ProductsController.cs file in the Controllers folder

```
...
[HttpPost]
public async Task<IActionResult>
        SaveProduct([FromBody] ProductBindingTarget target) {
    if (ModelState.IsValid) {
        Product p = target.ToProduct();
        await context.Products.AddAsync(p);
        await context.SaveChangesAsync();
        return Ok(p);
    }
    return BadRequest(ModelState);
}
...
```

The `ModelState` property is inherited from the `ControllerBase` class, and the `IsValid` property returns `true` if the model binding process has produced data that

meets the validation criteria. If the data received from the client is valid, then the action result from the `Ok` method is returned. If the data sent by the client fails the validation check, then the `IsValid` property will be `false`, and the action result from the `BadRequest` method is used instead. The `BadRequest` method accepts the object returned by the `ModelState` property, which is used to describe the validation errors to the client. (There is no standard way to describe validation errors, so the client may rely only on the 400 status code to determine that there is a problem.)

To test the validation, restart ASP.NET Core and use a new PowerShell command prompt to run the command shown in listing 19.26.

Listing 19.26 Testing validation

```
Invoke-WebRequest http://localhost:5000/api/products -Method POST -Body
➥(@{Name="Boot Laces"} | ConvertTo-Json) -ContentType "application/json"
```

The command will throw an exception that shows the web service has returned a 400 Bad Request response. Details of the validation errors are not shown because neither the `Invoke-WebRequest` command nor the `Invoke-RestMethod` command provides access to error response bodies. Although you can't see it, the body contains a JSON object that has properties for each data property that has failed validation, like this:

```
{
 "Price":["The field Price must be between 1 and 1000."],
 "CategoryId":["The field CategoryId must be between 1
    and 9.223372036854776E+18."],
 "SupplierId":["The field SupplierId must be between 1
    and 9.223372036854776E+18."]
}
```

You can see examples of working with validation messages in chapter 29 where the validation feature is described in detail.

19.5.5 Applying the API controller attribute

The `ApiController` attribute can be applied to web service controller classes to change the behavior of the model binding and validation features. The use of the `FromBody` attribute to select data from the request body and explicitly check the `ModelState.IsValid` property is not required in controllers that have been decorated with the `ApiController` attribute. Getting data from the body and validating data are required so commonly in web services that they are applied automatically when the attribute is used, restoring the focus of the code in the controller's action to dealing with the application features, as shown in listing 19.27.

Listing 19.27 The ProductsController.cs file in the Controllers folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;

namespace WebApp.Controllers {

    [ApiController]
```

```
    [Route("api/[controller]")]
    public class ProductsController : ControllerBase {
        private DataContext context;

        public ProductsController(DataContext ctx) {
            context = ctx;
        }

        [HttpGet]
        public IAsyncEnumerable<Product> GetProducts() {
            return context.Products.AsAsyncEnumerable();
        }

        [HttpGet("{id}")]
        public async Task<IActionResult> GetProduct(long id) {
            Product? p = await context.Products.FindAsync(id);
            if (p == null) {
                return NotFound();
            }
            return Ok(p);
        }

        [HttpPost]
        public async Task<IActionResult>
                SaveProduct(ProductBindingTarget target) {
            Product p = target.ToProduct();
            await context.Products.AddAsync(p);
            await context.SaveChangesAsync();
            return Ok(p);
        }

        [HttpPut]
        public async Task UpdateProduct(Product product) {
            context.Update(product);
            await context.SaveChangesAsync();
        }

        [HttpDelete("{id}")]
        public async Task DeleteProduct(long id) {
            context.Products.Remove(new Product() {
                ProductId = id, Name = string.Empty
            });
            await context.SaveChangesAsync();
        }

        [HttpGet("redirect")]
        public IActionResult Redirect() {
            return RedirectToAction(nameof(GetProduct), new { Id = 1 });
        }
    }
}
```

Using the `ApiController` attribute is optional, but it helps produce concise web service controllers.

19.5.6 Omitting Null properties

The final change I am going to make in this chapter is to remove the `null` values from the data returned by the web service. The data model classes contain navigation properties that are used by Entity Framework Core to associate related data in complex queries, as explained in chapter 20. For the simple queries that are performed in this chapter, no values are assigned to these navigation properties, which means that the client receives properties for which values are never going to be available. To see the problem, use a PowerShell command prompt to run the command shown in listing 19.28.

Listing 19.28 Sending a GET request

```
Invoke-WebRequest http://localhost:5000/api/products/1 |
➥Select-Object Content
```

The command sends a GET request and displays the body of the response from the web service, producing the following output:

```
Content
-------
{"productId":1,"name":"Green Kayak","price":275.00,
 "categoryId":1,"category":null,"supplierId":1,"supplier":null}
```

The request was handled by the `GetProduct` action method, and the `category` and `supplier` values in the response will always be `null` because the action doesn't ask Entity Framework Core to populate these properties.

PROJECTING SELECTED PROPERTIES

The first approach is to return just the properties that the client requires. This gives you complete control over each response, but it can become difficult to manage and confusing for client developers if each action returns a different set of values. Listing 19.29 shows how the `Product` object obtained from the database can be projected so that the navigation properties are omitted.

Listing 19.29 Omit properties in the ProductsController.cs file in the Controllers folder

```
...
[HttpGet("{id}")]
public async Task<IActionResult> GetProduct(long id) {
    Product? p = await context.Products.FindAsync(id);
    if (p == null) {
        return NotFound();
    }
    return Ok(new {
        p.ProductId, p.Name, p.Price, p.CategoryId, p.SupplierId
    });
}
...
```

The properties that the client requires are selected and added to an object that is passed to the `Ok` method. Restart ASP.NET Core and run the command from listing

19.28, and you will receive a response that omits the navigation properties and their `null` values, like this:

```
Content
-------
{"productId":1,"name":"Green Kayak","price":275.00,
 "categoryId":1,"supplierId":1}
```

CONFIGURING THE JSON SERIALIZER

The JSON serializer can be configured to omit properties when it serializes objects. One way to configure the serializer is with the `JsonIgnore` attribute, as shown in listing 19.30.

Listing 19.30 Configuring the serializer in the Product.cs file in the Models folder

```
using System.ComponentModel.DataAnnotations.Schema;
using System.Text.Json.Serialization;

namespace WebApp.Models {
    public class Product {

        public long ProductId { get; set; }

        public required string Name { get; set; }
        [Column(TypeName = "decimal(8, 2)")]
        public decimal Price { get; set; }

        public long CategoryId { get; set; }
        public Category? Category { get; set; }

        public long SupplierId { get; set; }

        [JsonIgnore(Condition = JsonIgnoreCondition.WhenWritingNull)]
        public Supplier? Supplier { get; set; }
    }
}
```

The `Condition` property is assigned a `JsonIgnoreCondition` value, as described in table 19.8.

Table 19.8 The values defined by the JsonIgnoreCondition enum

Name	Description
`Always`	The property will always be ignored when serializing an object.
`Never`	The property will always be included when serializing an object.
`WhenWritingDefault`	The property will be ignored if the value is `null` or the default value for the property type.
`WhenWritingNull`	The property will be ignored if the value is `null`.

The `JsonIgnore` attribute has been applied using the `WhenWritingNull` value, which means that the `Supplier` property will be ignored if its value is `null`. Listing 19.31 updates the controller to use the `Product` class directly in the `GetProduct` action method.

Listing 19.31 A model class in the ProductController.cs file in the Controllers folder

```
...
[HttpGet("{id}")]
public async Task<IActionResult> GetProduct(long id) {
    Product? p = await context.Products.FindAsync(id);
    if (p == null) {
        return NotFound();
    }
    return Ok(p);
}
...
```

Restart ASP.NET Core and run the command from listing 19.28, and you will receive a response that omits the `supplier` property, like this:

```
Content
-------
{"productId":1,"name":"Green Kayak","price":275.00,"categoryId":1,
    "category":null,
 "supplierId":1}
```

The attribute has to be applied to model classes and is useful when a small number of properties should be ignored, but this can be difficult to manage for more complex data models. A general policy can be defined for serialization using the options pattern, as shown in listing 19.32.

Listing 19.32 Configuring the serializer in the Program.cs file in the WebApp folder

```
using Microsoft.EntityFrameworkCore;
using WebApp.Models;
using Microsoft.AspNetCore.Mvc;
using System.Text.Json.Serialization;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<DataContext>(opts => {
    opts.UseSqlServer(builder.Configuration[
        "ConnectionStrings:ProductConnection"]);
    opts.EnableSensitiveDataLogging(true);
});

builder.Services.AddControllers();

builder.Services.Configure<JsonOptions>(opts => {
    opts.JsonSerializerOptions.DefaultIgnoreCondition
        = JsonIgnoreCondition.WhenWritingNull;
});
```

```
var app = builder.Build();

app.MapControllers();

app.MapGet("/", () => "Hello World!");

var context = app.Services.CreateScope().ServiceProvider
    .GetRequiredService<DataContext>();
SeedData.SeedDatabase(context);

app.Run();
```

The JSON serializer is configured using the `JsonSerializerOptions` property of the `JsonOptions` class, and `null` values are managed using the `DefaultIgnoreCondition` property, which is assigned one of the `JsonIgnoreCondition` values described in table 19.8. (The `Always` value does not make sense when using the options pattern and will cause an exception when ASP.NET Core is started.)

This configuration change affects all JSON responses and should be used with caution, especially if your data model classes use `null` values to impart information to the client. To see the effect of the change, restart ASP.NET Core and use a browser to request http://localhost:5000/api/products, which will produce the response shown in figure 19.10.

TIP The `JsonIgnore` attribute can be used to override the default policy, which is useful if you need to include `null` or default values for a particular property.

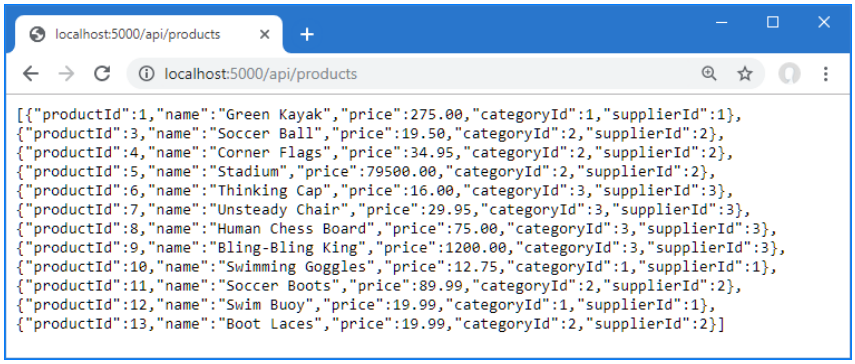

Figure 19.10 Configuring the JSON serializer

19.5.7 Applying a rate limit

In chapter 16, I demonstrated the rate limiting feature and showed you how it is applied to individual endpoints. This feature also works for controllers, using an attribute to select the rate limit that will be applied. In preparation, listing 19.33 defines a rate limiting policy and enables rate limits on controllers.

Listing 19.33 Rate limits in the Program.cs file in the WebApp folder

```
using Microsoft.EntityFrameworkCore;
using WebApp.Models;
using Microsoft.AspNetCore.Mvc;
using System.Text.Json.Serialization;
using Microsoft.AspNetCore.RateLimiting;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<DataContext>(opts => {
    opts.UseSqlServer(builder.Configuration[
        "ConnectionStrings:ProductConnection"]);
    opts.EnableSensitiveDataLogging(true);
});

builder.Services.AddControllers();

builder.Services.AddRateLimiter(opts => {
    opts.AddFixedWindowLimiter("fixedWindow", fixOpts => {
        fixOpts.PermitLimit = 1;
        fixOpts.QueueLimit = 0;
        fixOpts.Window = TimeSpan.FromSeconds(15);
    });
});

builder.Services.Configure<JsonOptions>(opts => {
    opts.JsonSerializerOptions.DefaultIgnoreCondition
        = JsonIgnoreCondition.WhenWritingNull;
});

var app = builder.Build();

app.UseRateLimiter();
app.MapControllers();

app.MapGet("/", () => "Hello World!");

var context = app.Services.CreateScope().ServiceProvider
    .GetRequiredService<DataContext>();
SeedData.SeedDatabase(context);

app.Run();
```

This listing sets up the same policy I used in chapter 16, which limits requests to one every 15 seconds with no queue. Listing 19.34 applies the policy to the controller using the `EnableRateLimiting` and `DisableRateLimiting` attributes.

> **NOTE** You can apply a single rate limiting policy to all controllers by calling `app.MapControllers().RequireRateLimiting("fixedWindow")`. This policy can be overridden for specific controllers and actions using the `EnableRateLimiting` and `DisableRateLimiting` attributes.

Listing 19.34 Limits in the ProductsController.cs file in the WebApp/Controllers folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using Microsoft.AspNetCore.RateLimiting;

namespace WebApp.Controllers {

    [ApiController]
    [Route("api/[controller]")]
    [EnableRateLimiting("fixedWindow")]
    public class ProductsController : ControllerBase {
        private DataContext context;

        public ProductsController(DataContext ctx) {
            context = ctx;
        }

        [HttpGet]
        public IAsyncEnumerable<Product> GetProducts() {
            return context.Products.AsAsyncEnumerable();
        }

        [HttpGet("{id}")]
        [DisableRateLimiting]
        public async Task<IActionResult> GetProduct(long id) {
            Product? p = await context.Products.FindAsync(id);
            if (p == null) {
                return NotFound();
            }
            return Ok(p);
        }

        // ...other action methods omitted for brevity...
    }
}
```

The `EnableRateLimiting` attribute is used to apply a rate limiting policy to the controller, specifying the name of the policy as an argument. This policy will apply to all of the action methods defined by the controller, except the `GetProduct` method, to which the `DisableRateLimiting` attribute has been applied and to which no limits will be enforced.

Restart ASP.NET Core and use a browser to request http://localhost:5000/api/products. Click the browser's reload button and you will exceed the request limit and see a 503 error, as shown in figure 19.11. You can request the URL http://localhost:5000/api/products/1 as often as you wish without producing an error.

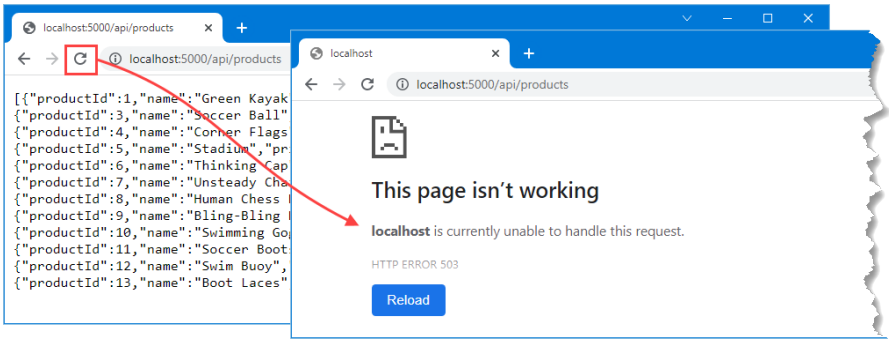

Figure 19.11 An error caused by a rate limit

Summary

- RESTful web services use the HTTP method and URL to specify an operation to perform.Web services can be created using top-level statements but using controller scales is better for most projects.
- The base class for controllers defines properties that access the request data.
- Action methods are decorated with attributes to specify the HTTP methods they accept.
- ASP.NET Core will perform model binding to extract data from the request and pass it to an action method as an object.
- Care must be taken to receive only the data that is required from the user.
- Data validation can be performed on the data that is produced by model binding, ensuring that clients provide data in a way the ASP.NET Core application can work with.
- The rate limiting features described in chapter 16 can also be applied to web services.